Lecture 19

Arrays Searching Sorting

An important topic in arrays is search an unordered data. The simple way is to start from first element, compare with target and continue with second, third, and till last value reached or target found. This is **brute force search**, **linear search** because we don't have any information about array values. The only choice is to check all values one by one. Obviously it doesn't matter whether we start from left or right in case of unordered data. Secondly we will discuss sorting which will be used in better search algorithms.

Search

Arrays can be used to store many values, greater than 50, 100, 200 is common but required some iterative technique to search instead of applying simple if checks. As we are discussing about general data which can be unordered, therefore, we will discuss only possible searching technique that is **brute force or linear search**:

```
public static boolean isExist(int n[], int target){
  int i;
  for (i=0;i<n.length;i++)
     if (target==n[i])
       return true;
  return false;
}
```

Above method is simply checking whether or not value exist in array. If value exist function will return true otherwise after termination of loop return false. Student should notice that writing else will be a big logical mistake because we cannot return false unless search is finished. Therefore, if all values are checked that means end of loop only than we can return false. Another mistake normally done by student is to print value because it has no use because we already sending target. There can be two reasons for search, either we want to see value exist or not or we want to perform some operation on target value. Like we may want to modify marks or address etc.

Above function is not much useful because in case value exists we don't know location. If we have to perform some operation we can't do that. Therefore a better function can be:

```
public static int indexOf(int n[], int target){
  int i;
  for (i=0;i<n.length;i++)
     if (target==n[i])
       return i;
  return -1;
}
```

Here we are using -1 as sentinel value because array has index 0 to onwards. No negative index is valid index. Therefore we may check for -1 as value not found; whereas; otherwise we will get index of target value in array and we are able to print as well as we can perform any operation like correcting typing error previously done.

Sorting

Sorting means to arrange value in some order either in ascending or descending. We will discuss in next lecture why we need ordering but for the moment trust my statement it is very useful activity to arrange in any order. We will discuss two sorting techniques here namely **Selection Sort** & **Bubble Sort**. Obviously both and more than dozen other techniques do sort values, the difference is of the way and time taken by each algorithm.

Selection Sort

Array having distinct values if sorted by any technique each value will have unique position. Objective of sorting is to place all values to their position in sorted array. In this technique we will select either largest or smallest value from selection and place it on its position. We will repeat the process for rest of the array and ultimately get the complete array sorted. Let us start from function to find position of max element from array:

```
public static int findMaxPosition(int n[]){
  int i, maxPosition =0;
  for (i=1;i<n.length;i++)
      if (n[maxPosition]<n[i])
        maxPosition=i;
  return maxPosition;
}
```

First of all student must notice that this function is different from search function. The fundamental difference is that search stops whenever we find element in the array; whereas; we cannot stop while locating max element until complete array is compared. This is the reason inside array we are just storing index of max element and outside array we are returning position. Unfortunately this function is also not very much helpful because each time we call this function it will return the location of max element of array; whereas; in sorting we are interested in next max or min elements position to adjust all elements. Therefore we will do a slight modification in above function that is adding a new parameter that is start of array. So that each time we can send a new starting position to get next position:

```
public static int findMinPosition(int n[], int start){
  int i, minPosition =start;
  for (i=start+1;i<n.length;i++)
      if (n[minPosition]>n[i])
        minPosition=i;
  return minPosition;
}
```

Here we have changed function to find min position in order to get sorting in ascending order. Also we place start in minPosition and start loop from start+1. Finally here we will use this function to write our selection sort function:

```
public static void selectionSort(int n[]){
  int i, minPosition, temp;
  for (i=0;i<n.length-1;i++){
    minPosition=findMinPosition(n,i);
    temp=n[minPosition];
    n[minPosition]=n[i];
    n[i]=temp;
  }
}
```

Hope many of you are clear about this algorithm but just a little explanation may clarify rest of you. Here we are running a loop from 0 (start of array) to n.length-1 (one lesser to the size of array). Each time we find position of minimum element and swap it with the element currently existing at the location. Just consider following array elements.

23 34 19 14 22

If we find position of min element, surely we will get 3 that is position of 14 [as position starts from 0]

Now we will replace 14 with 23 and we will get:

14 34 19 23 22

Now instead of finding min element again from same elements we will find min elements from n-1 elements on right side that is leaving 14. Surely now we will get 2 that is position of 19. Now we will replace 19 with 34 and get:

14 19 34 23 22

Going through this process one by one we will get complete array sorted.

Bubble Sort

Bubble sort is another technique to perform sorting. Comparatively this is simpler but relatively expensive technique that is more operation required to do same thing. To understand bubble sort just consider following code:

```
for (i=0;i<n.length-1;i++)
  if (n[i]>n[i+1]){
    temp=n[i];
    n[i]=n[i+1];
    n[i+1]=temp;
  }
```

This code will compare adjacent [neighboring] values and if values are out of order [that is greater value on left side], values will be swapped, again using same 3 step process. Going from 0 to length-1 and repeating same process ultimately we will get max element on right most position. Just consider following values:

23 34 19 14 22

After passing through this code, we will get:

23 19 14 22 34

Repeating same code for new values, we will get:

19 14 22 23 34

Therefore, repeating the process for length-1 times, we will get sorted array, hence just adding another loop over the given code, we will get bubble sort function:

```
public static void bubbleSort(int n[]){
  int i, j, temp;
  for (j=0;j<n.length-1;j++)
    for (i=0;i<n.length-1;i++)
      if (n[i]>n[i+1]){
        temp=n[i];
        n[i]=n[i+1];
        n[i+1]=temp;
      }
}
```

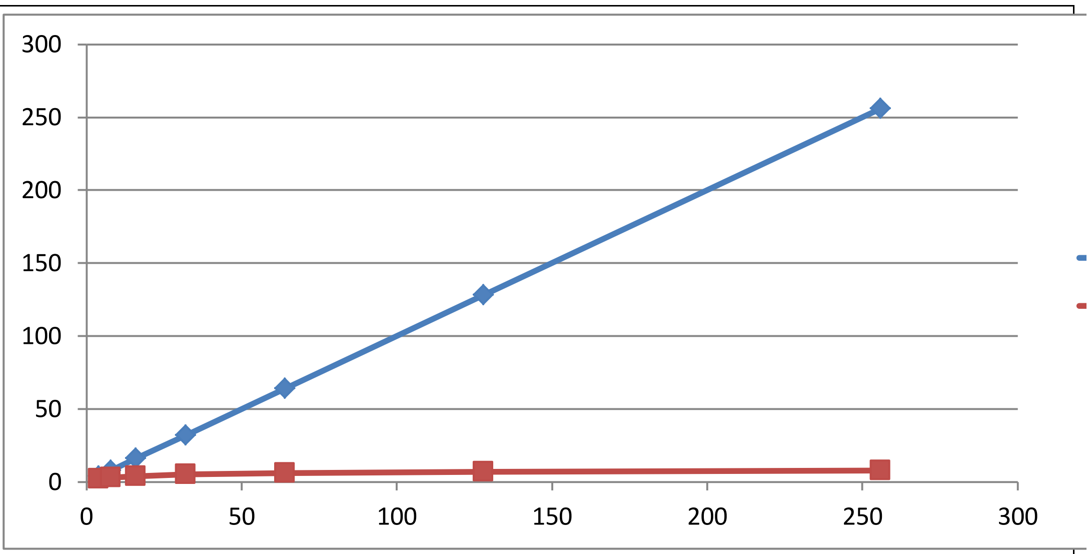
Hope like all students you will also feel that it is relatively simpler but important difference is that swap operations are inside inner loop so for array of size 10. We may have to perform swap operations 9+8+7+6+5+4+3+2+1=45 times; whereas in selection sort swap operation exist in outer loop and have to perform 9 times at max. This is huge difference. For general knowledge there exist many sorting algorithms like Insertion Sort, Merge Sort, Quick Sort, Bucket Sort, Hash Sort, Counting Sort. All of them do sorting accurately but have differences in terms of time besides mechanism. You may study more InshaAllah in your Data Structures course in next semester.

Important Note:

An important question is why to do sorting? Student's answer of this question is that we will get ordered data but do we have any use of ordered data besides showing data to user in some particular order. The most important use of data is to perform search. On unordered data we have no choice but to do a linear search that is to look all values in worst case; whereas; in case of ordered data we can perform binary search which in worst case requires comparison equal to the log of total values that is for 100 values we require 7 comparison and 8 if size is 200 and 9 comparisons for size 400 that is big difference between 400 comparisons and 9 comparisons.

The idea is if we have sorted data and we will start search from middle of the array we can easily judge whether to go left or right of the array. In any case we have to choose one side only. By repeating the same process that is to search in the middle of sub-array we can further reduce array. In first comparison we will left with half of the array, in second comparison we will left with quarter of the array and so on. This is how binary search proceeds. You can see the difference in values and in graph as well:

size	log(size)
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
2048	11
4096	12
8192	13
16384	14
32768	15
65536	16
131072	17
262144	18
524288	19
1048576	20



One may have question that sorting is also very expensive process comparing even linear search but issue is we don't have to perform sorting again and again if data is not changing abruptly; whereas; we may perform search many times.